02 BACKGROUND AND STATE OF THE ART

In this chapter we **TODO** Make introduction

2.1 WebAssembly

Starting with the Web 2.0 paradigm, JavaScript has long been the language for client-side scripting in modern web browsers. However, its complexity, security vulnerabilities, and performance limitations have led to the exploration of various alternatives for client-side scripting over the years [**?** **?** **?**]. Remarkably, these alternatives largely failed to gain traction, primarily due to security concerns and a lack of consensus among browser vendors.

In 2014, Alon Zakai and colleagues proposed Emscripten [**?**]. Emscripten used a strict subset of JavaScript, asm.js, to allow low-level code such as C to be compiled to JavaScript. This approach came with the benefits of having all the ahead-of-time optimizations from LLVM, gaining in performance on browser clients [**?**]. Asm.js was faster than JavaScript because it limited the language features to those that can be optimized in the LLVM pipeline. Besides, it removed the majority of the dynamic characteristics of the language. Since asm.js was a subset of JavaScript it was compatible with all engines at that moment. Asm.js demonstrated that client-code could be improved with the right language design and standardization.

Following the asm.js initiative, the W3C publicly announced the WebAssembly (Wasm) language in 2017. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017. The announcement of Wasm marked the first step into the standardization of bytecode in the web environment. Wasm is designed to be fast, portable, self-contained and secure, and it promises to outperform JavaScript execution [**?**]. Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

case of a full Minecraft version[2]; and the case of Blazor[3], a .Net virtual machine implemented in Wasm, able to execute C# code in the browser.

Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecode Alliance [**?**] proposed WebAssembly System Interface (WASI) [**?**]. WASI pionered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [**?**], making it suitable for edge-cloud computing platforms [**?** **?**].

2.1.2 Binary format

A WebAssembly binary usually consists of a single module that serves as the foundational unit for deployment, loading, and compilation. A WebAssembly program requires a host environment for execution. This host is tasked with loading the WebAssembly binary and allocating the necessary resources for its successful operation. The interaction between the engine and the WebAssembly program is fundamentally dictated by the structure of the binary itself. In essence, a WebAssembly binary serves as a contract between the WebAssembly program and the host engine. It comprises a mix of definitions and declarations that establish the rules of engagement. In this context, a declaration refers to a newly specified element, such as the bodies of functions inside the binary, while a definition refers to an element originating from the host, such as imported functions.

A Wasm binary is organized into sections, each with a distinct semantic role and specific placement within the module. This structured organization facilitates further machine code compilation. Table 2.1 provides a summary of these sections, detailing their ID, name, and purpose. As illustrated in Figure 2.1, a WebAssembly section is a byte sequence that begins with a 1-byte section ID, followed by an 8-byte section size, and concludes with the section content, which matches the previously indicated size.

Some sections, like the Start and Custom sections, are optional, enhancing the binary's flexibility. The Custom section, for instance, can store metadata such as the compiler used. The modular format allows for efficient parsing; a compiler can skip irrelevant sections, speeding up the compilation process. Additionally, only a few sections have order constraints, offering further flexibility. Custom sections can even be repeated multiple times within the binary.

[2] `https://satoshinm.github.io/NetCraft/`
[3] `https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor`

ID	Section Name	Description
0	Custom	Comprises two parts: the section name and arbitrary content. Primarily used for storing metadata, such as the compiler used to generate the binary.
1	Type	Contains the function signatures for functions declared or defined within the binary.
2	Import	Lists elements imported from the host, including functions, memories, globals, and tables.
3	Function	Details functions defined within the binary. Essentially maps Type section entries to Code section entries.
4	Table	Groups functions with identical signatures to control indirect calls.
5	Memory	Specifies the number and initial size of unmanaged linear memories.
6	Global	Defines global variables as managed memory for use and sharing between functions in the WebAssembly binary.
7	Export	Declares elements like functions, globals, memories, and tables for host engine access. The entry point of the WebAssembly binary is typically declared here.
8	Start	Designates a function to be called upon binary readiness, initializing the WebAssembly program state before executing any exported functions.
9	Element	Contains elements to initialize the binary tables.
10	Code	Contains the body of functions defined in the Function section. Each entry consists of local variables used and a list of instructions.
11	Data	Holds data for initializing unmanaged linear memory. Each entry specifies the offset and data to be placed in memory.
12	Data Count	Primarily used for validating the Data Section. If the segment count in the Data Section mismatches the Data Count, the binary is considered malformed.

Table 2.1: Overview of binary sections in WebAssembly binaries, following the WebAssembly 1.0 specification.

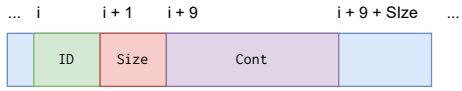


Figure 2.1: Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.

2.1.3 WebAssembly's Runtime structure

TODO Reorder this text according with the wrules

The WebAssembly runtime structure is described in the WebAssembly specification by enunciating 10 key components: the Store, Stack, Locals, Module Instances, Function Instances, Table Instances, Memory Instances, Global Instances, Export Instances, and Import Instances. These components are particularly significant in maintaining the state of a WebAssembly program during its execution. In the following text, we provide a brief description of each runtime component.

Store : The WebAssembly store represents the global state and is a collection of instances of functions, tables, memories, and globals. Each of these instances is uniquely identified by an address, which is usually represented as an i32 integer.

Module Instances : A module instance is a runtime representation of a loaded and initialized WebAssembly module. It contains the runtime representation of all the definitions within a module, including functions, tables, memories, and globals, as well as the module's exports and imports.

Table instances : A table instance is a vector of function elements. WebAssembly tables are used to support indirect function calls. For example, it allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm's compiler is in charge of populating the static table of functions.

Export Instances : Export instances represent the functions, tables, elements, globals or memories that are exported by a module.

Import Instances : Import instances represent the functions, tables, elements, globals or memories that are imported into a module from the host.

The Stack holds typed values and control frames, with control frames handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported.

Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

Memory Instances represent the unmanaged linear memory of a WebAssembly program, consisting of a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiates the WebAssembly module and the binary itself.

Global Instances : A global instance is a global variable with a value and a mutability flag, indicating whether the global can be modified or is immutable. Global variables are part of the managed data, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them.

Locals : Locals are mutable variables that are local to a specific function invocation. As globals, locals are part of the managed data.

Note 1. *Along with this dissertation, as the work of Lehmann et al. [**?**], we refer to managed and unmanaged data to differentiate between the data that is managed by the host engine and the data that is managed by the WebAssembly program respectively.*

Function Instances : A function instance is a closure, which is the pairing of a function's code with a module instance. This pairing is required because the function's code might refer to other definitions within the module instance, such as globals, tables, or memories. A function instance groups locals and a function body. Locals are typed variables that are local to a specific function invocation. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function.

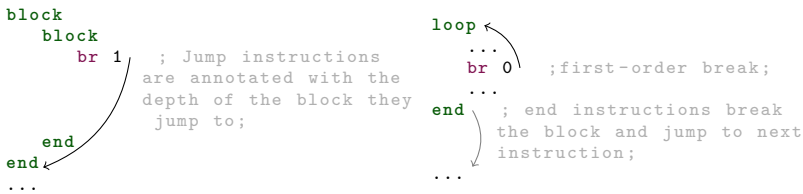
■ 2.1.4 Control flow

In WebAssembly, functions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, with instructions, where they start and end.

By design, each block operates autonomously and cannot reference or execute code from outer blocks. Control flow within a function is managed through three types of break instructions: unconditional break, conditional break, and table

break. Importantly, each break instruction is limited to jumping to one of its enclosing blocks.

Loops in Wasm are specialized blocks that can be restarted using a break instruction. Unlike standard blocks, where breaks jump to the end of the block, breaks within a loop block jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.1 provides an example comparing a standard block and a loop block in a Wasm function.



Listing 2.1: Example of breaking a block and a loop in WebAssembly.

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks.

For the purposes of this dissertation, we introduce a specific term to describe a particular kind of break within loops:

Definition 1. *Break instructions within loops that effectively jump to the loop's beginning are termed* first-order breaks.

2.1.5 From Source Code to Wasm

TODO Replace by Rust example **TODO** Annotate the Wasm code with the sections offset and length **TODO** Instantiate each one of the previously mentioned concepts

In Listing 2.2 and Listing 2.3 we illustrate a C program and the Wasm program that results from its compilation. The C function contains: heap allocation, external function declaration and the definition of a function with a loop, conditional branching, function calls and memory accesses. The code in Listing 2.3 shows the textual format for the generated Wasm. The module in this case first defines the signature of the functions (tpe1, tpe2 and tpe3) that help in the validation of the binary defining its parameter and result types. The information exchange between the host and the Wasm binary might be in two ways, exporting and importing functions, memory and globals to and from the host engine (import1, export1 and export2). The definition of the function (func1) and its body follows the last import declaration at import1.

```c
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi
                ↪ (12.54);
    }

    return A[0];
}
```

Listing 2.2: Example C function.

```
(module
  (type (;0;) (func (param f32) (
      ↪ result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
  (import "env" "ftoi" (func $ftoi (
      ↪ type 0)))
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block  ;label = @1;
      loop  ;label = @2;
        i32.const 0
        local.get 0
        i32.add
        i32.load
        local.tee 1
        i32.const 101
        i32.ge_s
        br_if 1 ;@1;
        local.get 0
        i32.const 4
        i32.add
        local.tee 0
        br_if 0 ;@2;
      end
      i32.const 0
      return
    end
    f32.const 0x1.9147aep+3
    call $ftoi
    local.get 1
    i32.add)
  (memory (;0;) 1)
  (global (;4;) i32 (i32.const 1000))
  (export "memory" (memory 0))
  (export "A" (global 2))
  (data $data (0) "\00\00\00\00...")
)
```

Listing 2.3: Wasm code for Listing 2.2.

The function body is composed of local-variable declarations and typed instructions that are evaluated using a virtual stack (Line 7 to Line 32 in Listing 2.3). Each instruction reads its operands from the stack and pushes back the result. The result of a function call is the top value of the stack at the end of the execution. In the case of Listing 2.3, the result value of the main function is the calculation of the last instruction, `i32.add` at result. As the listing shows, instructions are annotated with a numeric type.

2.1.6 WebAssembly's Ecosystem

TODO Go deep into the details of the tools that are comoing from parts of the jury

WebAssembly programs are pre-compiled from an array of source languages and are designed for execution in host environments such as web browsers. Though the execution of a WebAssembly program might be considered its final lifecycle stage, the WebAssembly ecosystem is far from simplistic. It comprises multiple stakeholders and a rich array of tools that cater to various needs. In the subsequent text, we describe the WebAssembly ecosystem by separating it into skateholders categories.

Producers , such as compilers, transform source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[4], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[5]. In parallel developments, the KMM framework[**?**] has incorporated WebAssembly as a compilation target, and the Javy approach[**?**] focuses on encapsulating JavaScript code within isolated WebAssembly binaries. This is achieved by porting both the engine and the source code into a secure WebAssembly environment. Blazor also enables the compilation of C code into WebAssembly binaries for browser execution[6]. Regardless of the source language or framework, the resulting WebAssembly binary functions similar to a traditional shared library, replete with code instructions, symbols, and exported functions.

Consumers encompass tools that undertake the tasks of validating, analyzing, transpiling to machine code, and executing WebAssembly binaries, e.g. browser clients. In the text that follows, we dissect them into specific categories and their respective domains of application.
TODO Locate WasmA [**?**]

Static and dynamic analysis, optimization and validation: The toolkit for analysing WebAssembly binaries is rich and varied. Tools such as Wassail[**?**], Wasmati[**?**], and Wasp[**?**] utilize a range of methodologies, including information flow control, code property graphs, and concolic execution, to identify vulnerabilities. Dynamic analysis counterparts like TaintAssembly[**?**], Wasabi[**?**], and Fuzzm[**?**] serve analogous functions. **TODO** Add veriwasm **TODO** Binanryen

Browsers Engines like V8[**?**], SpiderMonkey[**?**], and Chakra[**?**] are at the forefront of executing WebAssembly binaries. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code. For example, V8 employs quick, rudimentary optimizations such as constant

[4] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`
[5] A subset of the TypeScript language
[6] `https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor`

folding and dead code removal[7].

Standalone engines WebAssembly's applicability has transcended browser environments, largely due to the WebAssembly System Interface (WASI)[**?**]. WASI aims to standardize the interaction between host environments and WebAssembly modules, thereby facilitating portability of both bytecode and binaries across diverse platforms. It outlines a POSIX-like Application Binary Interface (ABI), which includes functionalities for filesystem access, network sockets, clocks, and random number generation, among others. Standalone engines such as WASM3[**?**], wasmtime[**?**], WAVM[**?**] and Sledge [**?**] have emerged to support WebAssembly. Similarly, Singh and colleagues [**?**] proposed a virtual machine for Wasm in Arduino based devices.

TODO Disect them into, JIT compilers and interpreters, ending with SWAM.

TODO Add the verifiable standalone engine here

Specialized Malware Detection In niche areas like cryptomalware detection, tools like MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] utilize static analysis techniques. Conversely, dynamic analysis is the forte of tools like SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**].

Smart Contract Analysis In the field of smart contracts, static analysis tools like EvalHunter[**?**], WANA[**?**], and EOSAFE[**?**] are employed to unearth vulnerabilities in WebAssembly-based contracts. Dynamic analysis tools in this sphere include EOSFuzzer[**?**] and wasai[**?**].

Obfuscation, and binary rewriting tools **TODO** WASMixer, Wobfuscator

While many existing tools purport to offer self-correctness evaluations and exhaustive test suites, the WebAssembly ecosystem is still in its early stages. To emphasize this, a 2021 study by Hilbig et al.[**?**] found a mere 8,000 unique WebAssembly binaries globally. This pales in comparison to mature ecosystems like JavaScript and Python, which offer 1.5 million and 1.7 million packages on npm[8] and PyPI[9], respectively. This limited pool of WebAssembly binaries presents a unique challenge for machine learning-based analysis tools, which require large datasets for effective training. This issue is explicitly addressed in one of the contributions of this dissertation[**?**]. Additionally, the scarcity of WebAssembly programs exacerbates the problem of software monoculture, increasing the likelihood of consuming a compromised WebAssembly program[**?**]. Besides, the current size of the WebAssembly ecosystem offers a small testing environment for evaluating the capabilities of consumer tools. This dissertation

[7] This analysis was corroborated through discussions with the V8 development team and through empirical studies in one of our contributions[**?**]

[8] `https://www.npmjs.com/`

[9] `https://pypi.org/`

aims to address these issues by introducing a comprehensive suite of tools. These tools are crafted to not only bolster the security of WebAssembly programs through Software Diversification but also to intensify the testing rigor for both producers and consumers in the ecosystem.

To the best of our knowledge, the most complete survey about Wasm tooling is presented in the Awesome Wasm (`https://github.com/mbasso/awesome-wasm`) repo. It is a cumulative GitHub repo which includes references to articles, papers, books, demos, compilers and engines related to Wasm.

2.1.7 WebAssembly's Security

TODO Check the attack primitives defined by Lehmann.

As we described, Wasm is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [**?**] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, implementations in both browsers and standalone runtimes [**?**] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [**?**]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [**?**] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap by overflowing the stack. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for Wasm. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [**?**]. Several proposals for extending Wasm in the current roadmap could address some existing vulnerabilities. For example, having multiple memories[10] could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors[11].

[10] `https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md`

[11] `https://webassembly.org/roadmap/`

2.2 Software diversification

- 2.2.2 Generating Software Diversification
- 2.2.3 Variants generation
- 2.2.4 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. `https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13`

2.3 Exploiting Software Diversification

- 2.3.2 Defensive Diversification
- 2.3.3 Offensive Diversification